WPI-CS-TR-17-02 November 2017

The Impact of a Single Lecture on Program Plans in First-Year CS

by

Francisco Enrique Vicente Castro
Shriram Krishnamurthi
Kathi Fisler

The Impact of a Single Lecture on Program Plans in First-Year CS (Technical Report Version)[*]

Francisco Enrique Vicente Castro
WPI Dept of Computer Science
fgcastro@cs.wpi.edu

Shriram Krishnamurthi, Kathi Fisler
Brown Dept of Computer Science
{sk,kfisler}@cs.brown.edu

ABSTRACT

Most programming problems have multiple viable solutions that organize the underlying problem's tasks in fundamentally different ways. Which organizations (a.k.a. *plans*) students implement and value depends on the solutions they have seen before as well as features of their programming language. Many first-year courses teach different equivalent low-level constructs (such as `for` vs `while` loops) without also discussing different higher-level plans. How much exposure to planning do students need before they can appreciate and produce different solution plans?

We report on a study in which students in two introductory courses at different universities were given a single lecture on planning between assessments. Surprisingly, that one lecture sufficed to get students to produce multiple high-level plans (including ones first introduced in the lecture) and to richly discuss tradeoffs between plans. This suggests that planning can be taught with fairly low overhead once students have a reasonable foundation in programming.

KEYWORDS

Plan composition, program design, programming pedagogy

1 INTRODUCTION

Given a programming problem, students make multiple choices in crafting a solution. Some choices focus on lower-level concerns such as which language constructs to use (e.g., a `while` loop versus a `for` loop). Higher-level decisions include how to cluster the subtasks of a problem into individual functions or code blocks. The clustering of subtasks is often called a *plan* [19]. Programming involves (among other things) implementing plan components in lower-level constructs and composing those constructs into a solution for the overall problem.

For students who will study computing further, dealing with planning is instructive. Arguably, the ability to imagine multiple plans for the same problem is a powerful and useful programming skill. Plans have tradeoffs: some are tuned for efficiency, some adapt better to changing requirements, some are easier for humans to maintain, and so on. Computer science students should, at some point, learn to see problems through plans, and be able to envision and implement multiple plans while weighing their tradeoffs. When this should happen and how much instruction this requires are, however, open questions.

However, planning is not an advanced topic only for upper-level students or students who intend to work as software engineers. Even casual programmers who write scripts more than full-fledged software are affected by planning decisions. A science student writing scripts to process data for a lab experiment encounters changing data requirements, noisy data, or other situations that get handled through planning, not just low-level construct choices. Thus, planning is relevant even to students in first-year computing courses, including those who may not take later courses.

But can we teach planning that early, and if so, how? Some researchers have made notable efforts at teaching students high-level strategies, patterns, or plans from the outset [4, 8], but this requires a concerted effort to make planning a central part of the introductory curriculum. Because these courses serve a variety of needs for the rest of the program, doing so requires buy-in and prioritization from all the other faculty, who may have vastly different disciplinary concerns. Thus, we were interested in whether a lightweight approach to teaching planning could have any effect, or whether only a comprehensive overhaul of the courses—which may be impossible given a department's other needs—would suffice.

Concretely, we assigned students programming problems—of the form the courses anyway expected them to become comfortable writing—that could be approached through multiple plans; leveraged those to give a single 50-minute lecture on plans and tradeoffs; and then assigned a new set of programming problems. In the post-assessment, we asked students to produce two solutions to each problem, each embodying a different plan. We also asked students to rank their solutions by preference, so we could see what criteria they were learning to apply to plans. We ran the experiment in two courses (at different universities, with slightly different contexts), neither of which had discussed plans, high-level design choices, or efficiency concerns prior to the experiment.

We expected that many (if not most) students would produce two solutions with different low-level constructs, but the same high-level plans. We were pleasantly surprised that nearly all students did produce two different plans in the post-assessment, often choosing a general plan structure that was first introduced in the planning lecture. When asked to preference-rank among their solutions, many students chose solutions with a different general structure than what they wrote on the pre-assessment. In one of the courses, we gathered pre-lecture data on the criteria students used to rank different solutions; these students raised considerably more varied criteria when ranking the post-assessment solutions. All in all, these results suggest that lightweight instruction in planning can have signficant impact, assuming students can produce a single correct solution for the pre-assessment questions.

[*]This version can be cited as WPI Dept of Computer Science Technical Report number WPI-CS-TR-17-02, released November 2017.

Beyond this study, another contribution of this work lies in the methodology that it proposes for studying planning. Plan-composition studies seem to be having a resurgence [2, 16], but it is time the field moved beyond looking just at errors to taking a more holistic look at students' planning choices and practices (which in turn could better explain some of the errors observed in prior studies). We therefore hope this paper will contribute to a larger discussion about research directions in program planning for both majors and non-majors.

2 RELATED WORK

The task of developing and integrating programming plans has been identified as a recurring problem among programming students [3, 18, 19]. The landscape of first-year programming courses tends to focus instruction on low-level programming constructs, with the expectation that students implicitly build their knowledge base of problem solving and programming strategies from trial-and-error through extensive sets of exercises [3]. While some recent studies show students succeeding at plan composition in specific contexts [6, 16], the pedagogic choices that help students with this task remain poorly understood.

A growing body of research aims at improving planning skills through pedagogical frameworks and practices that explicitly teach systematic problem solving strategies in programming. Porter and Calder draw on the concept of programming *patterns*, known solutions used for recurring design or programming problems: their work suggests a process for building a pattern vocabulary for guiding students through problem decomposition [14]. Muller, Haberman, and Ginat used this same concept in developing pattern-oriented instruction [10]: this approach involves attaching labels to algorithmic patterns and presenting various problems to students, while encouraging students to look for common patterns across problems. Students in the latter effort were successful at applying the patterns at the end of the course. Our work differs in trying a lightweight approach, in which planning is the focus of a lecture and two assignments rather than the entire curriculum.

De Raadt, Watson, and Toleman, in incorporating programming strategies in an introductory programming curriculum, make the explicit distinction between programming *knowledge* (language syntax and semantics) and programming *strategies* [4]. They include a 'strategy guide' that discusses *abutment*, *nesting*, and *merging* as ways for integrating strategies; their assignments require students to apply specific strategies in their solutions. Our work, in contrast, focuses on *problem-level techniques* (such as cleaning data) rather than code-level techniques (such as merging code). Unlike all three works, we also ask students to discuss design tradeoffs, as a way of triangulating what they understand about patterns. Arguably, the skill of evaluating tradeoffs becomes valuable as students progress through computing courses; as the contexts in which students solve problems grow more subtle, design tradeoffs take on greater importance.

3 COGNITIVE FOUNDATIONS OF RECALLING SOLUTIONS

Different strategies for teaching planning build on results of how people construct programs at a cognitive level. Given a programming problem, programmers (subconsciously) identify solutions to similar problems and adapt them to the constraints of the problem at hand [12, 13, 15, 20]. Repeated application of a pattern helps programmers form a mental schema for that problem (which could be recalled later for solving other problems); repeated use of a schema strengthens later recall of that schema [9, 11]. This basic architecture underlies curricular approaches to teaching patterns explicitly (as reviewed in section 2).

How much exposure do students need to a solution schema before they can apply it to new problems? This is an open question, and one that depends on a student's experience level. We would not expect a truly novice programmer to internalize a solution that used constructs (such as iteration) that the student had simply been shown in class: internalizing code patterns requires practice with actually using them. But what if a student had written several programs that traverse lists (for example), then saw a program that traverses a list to accomplish a slightly different goal than before? It seems plausible that the student could subsequently produce a program that handles the latter goal, even without separate practice (by virtue of having internalized both list traversal and any other constructs required for the latter goal). Given that there are differences between knowledge schemas and strategy schemas (as both de Raadt *et al.* and Caspersen cite [1, 4]), students may require less direct practice to internalize a new pattern that built on already-internalized schemas.

These results frame our experiment because our lightweight planning lecture shows students new ways to cluster subtasks of planning problems. All of our participants had been learning and practicting writing list traversals prior to the experiment (naturally, some were better at this than others). The planning lecture showed new (relative to the course contents) high-level ways to decompose a planning problem into (potentially multiple) list traversals. Our study asks whether students would apply these high-level strategies to new problems based just on the single lecture (without us telling them which solution style to produce, as other pattern-based studies have done [4]).

Of course, failure of a student to apply the new strategies could be due to many factors: students might not have understood the new planning patterns, they might have understood them but not liked them, they might not have had enough practice to use them confidently, and so on. The various components of our study attempt to tease out some of these factors, as the study design in section 4 describes.

4 RESEARCH QUESTIONS

Building on the cognitive literature described in section 3, our project explores the following research questions:

- *Can planning be taught at all in the first year? Or is it a topic that can only be covered after students have had significant experience with programming, software engineering, and/or computer science?*

- *Assuming it can be taught, what are the differences between groups of students in their ability to construct multiple plans, rank different plans, and talk about programs with a plan-oriented vocabulary?*
- *Assuming students can engage in these plan-oriented activities, how much of an intervention is necessary before they can do so? Does the class need to be restructured to make planning a focus, or can it be done with a lightweight intervention?*

On questions about the feasibility of a lightweight treatment of planning in the first year, our answers are a qualified "yes". However, our work raises as many questions as it answers; therefore, throughout the paper we list natural areas for follow-up. These are indicated by ☞ followed by a number. The discussions are all in section 8, where the number indicates which entry discusses it.

We show that:

- Planning can be covered in the first year for at least some populations of students.
- We do see quantitative differences between students based on prior programming experience [☞1].
- These results were obtained with a rather lightweight intervention: students wrote the kinds of programs they anyway would have in the course, and only one class lecture was devoted to the topic of planning. This suggests that planning is a topic that can be successfully integrated into a variety of courses without having to restructure the courses around planning.
- However, the students in our courses did have the benefit of studying functional programming ("functional" in the sense of languages like Haskell or Scheme, not C), so it is unclear whether a similar intervention would work in a traditional, imperative programming class [☞2]. Thus, this raises interesting questions about how we approach teaching introductory programming.

In what follows, we first describe our context and intervention (section 5.1), followed by a detailed analysis of our findings (section 6) that elaborates on these points.

5 STUDY DESIGN

At a high level our study had three components, which we applied slightly differently in each of two courses (section 5.1). Section 5.2 and section 5.4 discuss the specific problems used on the assessments. The three components were:

(1) A pre-assessment (section 5.2) in which students were asked to produce solutions to 2–3 programming problems. In one course, which had more time for this, students were also given 2–3 solutions to different problems and asked to rank them (with justification) in order of their preference between these solutions.
(2) A single 50-minute lecture (section 5.3) on planning and design tradeoffs. The lecture used the pre-assessment problems to frame the discussion, showing different high-level ways to decompose a planning problem into a collection of list traversals. The same professor gave the same lecture in both courses.
(3) A post-assessment (section 5.4) in which students were asked to (a) produce two solutions with different plans to each of 4 programming problems and (b) to preference rank between their solutions (with justification).

Ideally, we would have asked students to write multiple solutions with different structures in both the pre- and post-assessments. This would have let us gauge whether students could construct different plans even before the lecture, and also given us insight into which planning strategies students already knew. Unfortunately, we were unable to find a way to phrase this task that was not either extremely frustrating to students (because they could not understand the required task) or that did not essentially give away the answer. We still gain some insight into their background from their rankings (section 6.2), but determining how to establish a baseline more authoritatively remains an open question [☞3].

The questions in the pre- and post-assessment were carefully chosen to introduce some broadly-applicable strategies in multi-task programming problems. In particular, we exposed students to problems with the following features:

- Noisy data that could be *cleaned* prior to performing the main computation.
- Flattened data that could be *parsed* (or *reshaped*) to a structure that was better suited to the main computation.
- Overly-long data that could be *truncated* to a prefix of interest for the main computation.

In addition, the posttest included computations that targeted a *projection* of the data (say to a specific field within an object). We did not emphasize projection in the pre-assessment as students had experience with this idea from other assignments in both courses.

We used the lecture to discuss cleaning, parsing, and truncating in the context of the pre-assessment problems. We also discussed various design tradeoffs that these offered, including impact on run-time efficiency, ability to adapt the solution to a different dataset, and readability and maintainability of the resulting code.

5.1 The Host Courses

We conducted the study in two first-year CS courses at different universities. Each course was taught by one of the authors of this paper. Students in both courses had some prior programming experience, but the nature of that experience differed both across and within the populations. We describe each course in turn.

- CrsA is an acclerated CS1 course that compresses much of the first year into a semester. Students test into the course after one month in the department's regular CS1 course. Though it is open to all, most students in the course have some prior experience, usually with imperative or object-oriented programming in Java or Python. The course is taught in Pyret, a functional language with syntax reminiscent of Python.
- CrsB is a CS2 course on object-oriented programming and data structures, taught in Java. Students feed into the course from one of two introductory courses taught in functional programming: one feeder (CrsBnvc) course is for novice programmers, while the other (CrsBexp) is for students with prior programming experience. Students from CrsBnvc have seen little to no imperative programming prior to CrsB, while students from CrsBexp have prior experience similar to that of CrsA.

These descriptions indicate that we actually have three student populations within our two courses, with interesting overlaps among them. This table summarizes these populations:

Course	Course Lang	Prior Exprnc.	Prior Imprtv.	Used Iterators
CrsA	Pyret	maybe more	yes	yes
CrsBnvc	Java	7 weeks	no	a bit
CrsBexp	Java	maybe more	yes	yes

"Prior Exprnc." estimates students' programming experience prior to the current course: courses are 7 weeks long at the CrsB institution, while most students in CrsA and CrsBexp audiences may have had a year or more of prior programming. "Prior Imprtv." indicates whether students had previously programmed imperatively. "Used Iterators" says how much students had worked with higher-order functions (such as `map` and `filter`) before the pre-assessment. These constructs featured heavily in CrsBexp and CrsA, but were introduced more lightly in CrsBnvc.

Both the CS1 course that preceded CrsB and the CS1 course from which students placed into CrsA followed a similar program design curriculum [5], taught in functional programming with Racket. While the overall assignments and lectures in these two CS1 courses were not identical, they taught largely the same concepts and the same method for developing programs and writing good test suites. While we cannot control for differences in the pre-university programming background of students participating in this study, the common CS1 foundations provide some degree of a shared baseline.

Prior to the pre-assessment, CrsB had covered both kinds of `for` loops for iterating over Java linked lists. In-class examples of `for`-loops consisted of simple list traversals that accumulated answers (such as summing a given field across a list of objects) or filtering out a subset of elements. The pre-assessment was the first assignment in the course on programming with lists and `for`-loops.

Sampled Populations. All in all, there were 75 students in CrsA and 290 in CrsB. While all students completed the study, our (manual) analysis uses a sample based on final course grade. Acknowledging that overall course performance (as indicated by formal course grade) could be a relevant factor, we aimed for a sample of 10 students from each passing grade (A, B, and C) in each course. Since CrsB had two different feeder populations, we sampled separately from both feeder populations. Some subpopulations had fewer than 10 students in a grade band who submitted both assessments working individually (CrsB allowed pair work). We had seven C-range students in each course, eight B-range students in CrsBexp, and a full 10 students in each other population. In total, our sample included 27 CrsA students, 27 CrsBnvc students, and 18 CrsBexp students. The different grade bands did not manifest meaningfully in our analysis, so we do not discuss them further.

5.2 Pre-Assessment

The pre-assessments for both courses contained 2 or 3 programming problems; in CrsA, students were also asked to preference rank among solutions to 3 additional problems. For the pre-assessment, we used the problems from Fisler *et al.*'s recent study [7], as they had been designed for multi-linguistic contexts such as ours. We reproduce here the statements of problems that feature in our analysis, but defer descriptions of the other study problems to their paper or to our study handouts (complete with problem statements and the solutions given in the ranking questions), which are available online.[1]

In CrsA, the pre-assessment consisted of the *Palindrome*, *Sum Over Table*, and *Adding Machine* problems. Our analysis looks at *Adding Machine*:

> Design a program called `addingMachine` that consumes a list of numbers and produces a list of the sums of each non-empty sublist separated by zeros. Ignore input elements that occur after the first occurrence of two consecutive zeros.

Adding Machine features flattened data that could be parsed into a list of sublists and a prefix of data (prior to the consecutive zeros) that could be truncated. Typical solution structures include:

Single Traversal Traverses the input data once, accumulating (a) the sum of the current sublist and (b) the output, returning the output when consecutive 0s are detected.

Nested Traversal Like Single Traversal, but an extra inner loop re-traverses the sublists to compute their sums.

Parse One traversal converts the pre-00 input into a list of sublists; a second traversal produces the list of sums of each sublist.

Clean One traversal truncates the data prior to 00; subsequent traversals follow one of the other solution structures.

We did not give the same programming problems in CrsB because the instructor felt they were beyond what the students were prepared to do. Students did not know the string- or array-operations needed for *Palindrome*. The consecutive-position delimiter in *Adding Machine* would also have been new to the students from CrsBnvc. For CrsB, we instead used two different problems from the Fisler *et al.* study, specifically the classic *Rainfall* problem and *Length of Triples*. *Rainfall* involves noisy data (negative numbers that should not be averaged) and a single-character delimiter for the relevant data. *Length of Triples* asks for the longest concatenation of three consecutive elements from a list of strings.

For the questions that asked students to rank solutions in CrsA, students were given multiple solutions to three programming problems, asked to state their preferences among the solutions, and told to justify their decision (we did not suggest criteria for the comparison). For this component, we used the same ranking problems as in the Fisler *et al.* paper [7] (their paper includes a link to their detailed problem statements; we simply adapted their solutions to the programming languages used in our courses). The specific problems were the *Rainfall* and *Length of Triples* problems given to the CrsB students as programming problems, as well as *Shopping Cart* problem that asked students to compute the total cost of a shopping cart after applying discounts when certain volumes of items were being purchased.

We did not include a ranking problems portion in CrsB due to time constraints within that course. The programming problems were given within a larger assignment in that course, and the instructor did not feel the students could handle the additional work of the ranking problems in the time available for the assignment.

5.3 The Lecture Intervention

Within two days after the pre-assessment was due, Fisler lectured about the problems in each course (guest lecturinging in the course

[1] https://github.com/franciscastro/koli-2017

for which she was not the regular instructor). The lectures were not identical since the pre-assessment questions were different, but they covered similar content.

In CrsA, the lecture started with a discussion of the ranking problems and the tradeoffs students considered. Fisler moderated discussion among the students, making sure that each of efficiency, aesthetics, maintainability, and code structure were given due attention. The instructor showed possible solutions to *Adding Machine*, explicitly discussing parsing and truncating as applicable strategies.

In CrsB, Fisler showed multiple solutions to each of *Rainfall* and *Length of Triples*. The former was used to point out cleaning and truncating as strategies; the latter was used to point out parsing. These solutions were posted for later reference. Again the instructor moderated a classwide discussion among the students of the tradeoffs across these solutions to both problems.

In both lectures, Fisler described planning as the general task of allocating subproblems to traversals of the data. While this is a somewhat more code-focused definition that we might otherwise like, it was designed to give students a way to assess whether their two solutions would be "different" from the perspective of the post-assessment, though at this point they did not know what they would be asked to do on that assignment. However, the emphasis of the lecture was not on this definition but on concrete strategies, to help them build a vocabulary of planning operations.

5.4 Post-Assessment

For the post-assessment, we sought problems that were amenable to the parsing, cleaning, and truncating strategies discussed in the lecture. We wanted problems that resembled, but were not identical to, the pre-assessment problems. The post-assessment contained four problems; students were required to (a) submit two solutions (with different structures) for each problem, and (b) state a preference between their two solutions (with justification). Teaching assistants in both courses were instructed not to give students much help in figuring out what a second solution would look like, but rather to refer them to their notes from each lecture.

This paper focuses on two of the problems, due to their particular similarities with pre-assessment problems:

Data Smoothing Given a list of health records with a numeric `heartRate` field, design a program `dataSmooth` that produces a list of the `heartRate`s but with each (internal) element replaced with the average of that element and its predecessor and successor. E.g., given a list of health-records with heart rates `[95, 102, 98, 88, 105]`, the resulting smoothed sequence should be

```
    [95, 98.33, 96, 97, 105]
```

This problem, like the *Length of Triples* problem which both courses saw in the pre-assessment, looks at three-element windows within an input list. This problem is a good candidate for parsing. Other viable strategies include first extracting all the heart-rates from the health records (resulting in a list of numbers to smooth), or simply doing the entire computation in a single traversal of the input data.

Earthquake Monitor Write a program that takes a month and a list of readings from an earthquake sensor. In the input, 8-digit numbers are dates and numbers below 500 are readings for the preceding date. Produce a list of reports containing the highest reading for each date in the given month. E.g., given this list and 10 for the month,

20151004 200 150 175 20151005 0.002 0.03 20151207

the program should yield `[report(4, 200), report(5, .03)]`

This problem resembles *Adding Machine* in having sublists within the data, which makes it a candidate for parsing. Like both *Adding Machine* and *Rainfall*, it has a sentinel (in the form of data from a later month). It could be approached with a single traversal that accumulates the max value per date, a cleaning phase that restricts the input data to the desired month, or parsing prior to computing the reports.

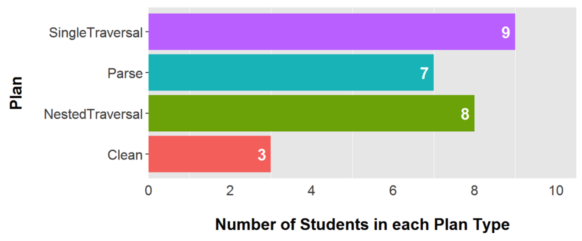

Figure 1: Structures of *Adding Machine* solutions in CrsA from the pre-assessment

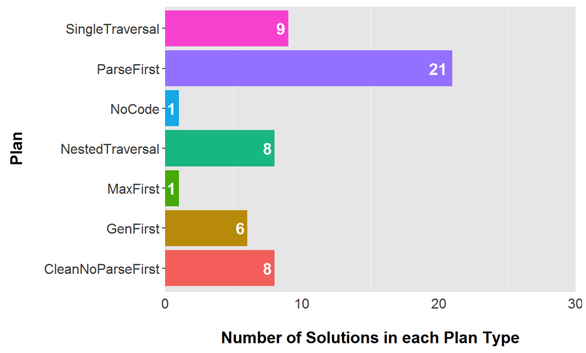

Figure 2: Structures of *Earthquake Monitor* solutions in CrsA from the post-assessment

6 ANALYSIS

We now discuss our findings. We find it useful to organize our results by course, first discussing what we observe from CrsA, then contrasting those results to the data from CrsB. A direct comparison of the results from the two courses is not meaningful due to differences in the students' backgrounds and the smaller set of questions used with CrsB. We still find value, however, in seeing how two groups of students with some similarities in their backgrounds fared in this experiment. The analysis will show interesting differences across the two courses; this gives a much more realistic picture of the proposed intervention than if we had reported on one course alone, despite the differences in the details of problem selection.

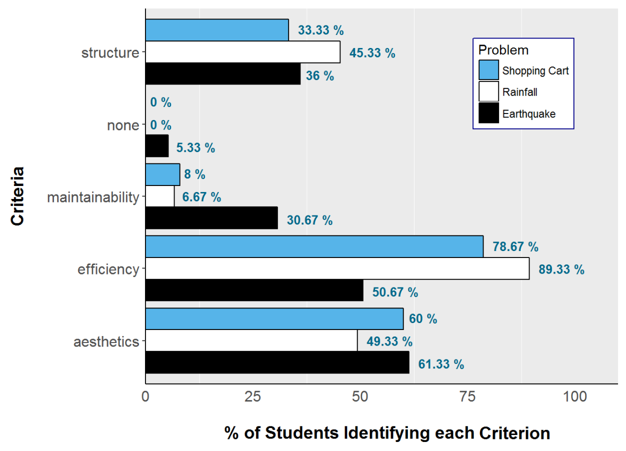

Figure 3: Criteria CrsA **students cited while ranking solutions to *Rainfall* (pre), *Shopping Cart* (pre), and *Earthquake Monitor* (post)**

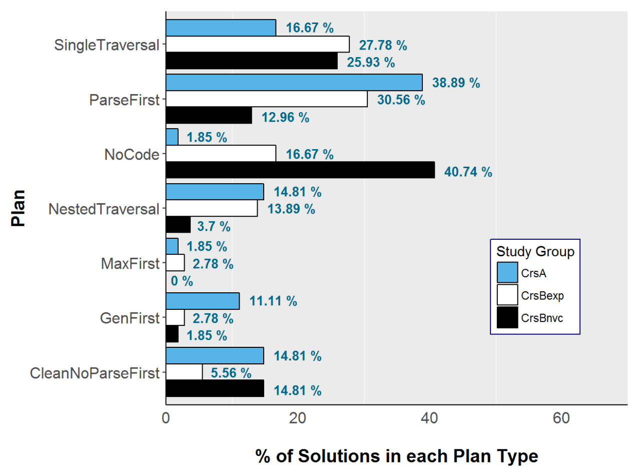

Figure 4: Structures of *Earthquake Monitor* solutions from the post-assessment

6.1 The View from CrsA

The data from CrsA suggest that our intervention lecture had a noticeable impact on students' planning behavior. Figure 1 and fig. 2 show the structures that CrsA students used in *Adding Machine* on the pre-assessment and *Earthquake Monitor* on the post-assessment, respectively (appendix A outlines how we coded the solutions). We contrast these two problems because they have similar attributes: flattened sequences of structured data, with a computation (sum or max) to be performed on a delimited subsequence of relevant data. Students took a variety of approaches in the pre-assessment, with some using parsing. Usage of parsing jumps significantly ($p < .006$ with a McNemar's test) in the post-assessment: about 70% of CrsA students used parsing in one of their two *Earthquake Monitor* solutions. Even in *Data Smoothing*, roughly the same number of students chose to parse as did a single data traversal. Thus, there is evidence that CrsA students learned and applied the parsing strategy from the single lecture.

Significant contrasts also arise when we examine CrsA students' ranking preferences between the two assessments (appendix B outlines how we coded the solutions). Figure 3 shows the criteria that students used when ranking the *Rainfall* and *Shopping Cart* problems from the pre-assessment, alongside those for *Earthquake Monitor* on the post-assessment (these are for the entire population of 75 students, not just the sampled subset). The three most common categories (efficiency, structure, and aesthetics) are the same in the two pre-assessment problems, though aesthetics is more prominent on *Shopping Cart*, which involves more subtasks than *Rainfall*. In the post-assessment, efficiency is cited significantly less often ($p < .0001$) while maintainability grows significantly ($p = .02$). Maintainability is a potential issue for both *Rainfall* and *Shopping Cart*, but particularly the latter (as the store could begin to offer more or different discounts). Table 1 shows the evolution of these criteria on a per-student (rather than aggregate) level, summarizing numbers of CrsA students who raised various criteria in each of the assessments. The table shows that many students both dropped and gained criteria over the course of the study.

Criterion	Pre, not post	Post, not pre	Pre and post
Efficiency	31	-	36
Structure	24	9	15
Aesthetics	14	1	35
Maintainability	7	20	3

Table 1: Criteria raised by CrsA **students across pre- and post-assessment rankings**

All in all, the lecture had the impact we hoped for in CrsA: students showed their ability to produce solutions with multiple plans (only 4 students had the same high-level plan on *Earthquake Monitor* and only 5 had the same high-level plan on *Data Smoothing*; only one student overlaps these two groups), most students raised more issues when discussing tradeoffs among solutions, and many students changed the solution structures that they preferred in the post-assessment (which is merely a sign that the lecture impacted their thinking, not that their analyses necessarily grew more accurate [☞4]).

6.2 The View from CrsB

Contrasting the CrsA data with those from CrsB paints a more nuanced picture of the impact of the single lecture. In particular, taken as a whole, the lecture's impact is less significant; we also see interesting differences between the CrsBnvc and CrsBexp subgroups of CrsB. We also see differences between CrsA and CrsBexp, who had been working in different programming languages despite a fairly common curriculum (and common programming language) just a month or two prior to the study.

Figure 4 contrasts the *Earthquake Monitor* solutions across all three populations in the post-assessment. Two observations jump

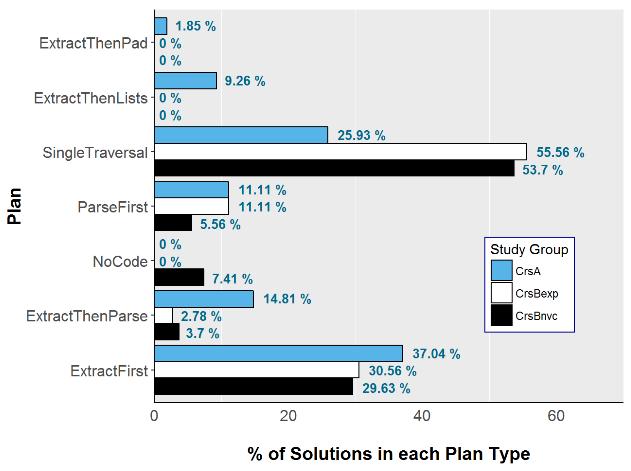

Figure 5: Structures of *Data Smoothing* solutions from the post-assessment

out. First, a significant percentage of students in CrsB were unable to solve the problem at all (the *"No Code"* group): of the 45 students sampled, 11 turned in no solution for *Earthquake Monitor* (9 from CrsBnvc, 2 from CrsBexp), while another 6 students turned in only one solution. Of those who turned in only one solution, half used parsing while the others did a straightforward loop-based traversal or nested traversal. In contrast, there was only one case of a *"No Code"* solution in CrsA. We suspect that the *"No Code"* s came partly from the lack of programming experience in CrsBnvc and partly from students running out of time (*Earthquake Monitor* was the last problem on the assignment, which was due just before students left campus for Thanksgiving, a mid-course holiday.)

Setting aside the *"No Code"* students, the dominant solution structures differ across the three populations: *"Parse First"* dominates in CrsA, *"Single Traversal"* dominates in CrsBnvc, while these two are fairly even in CrsBexp. This suggests that parsing strategies may be harder for students to adopt with only novice programming experience ([☞5]).

On *Data Smoothing* (fig. 5), the two CrsB populations are more similar to each other, with much heavier use of *"Single Traversal"* solutions, especially compared to the dominance of *"ExtractFirst"* solutions in CrsA. Here, we strongly suspect that programming language constructs were a factor. Most of the CrsA students used a built-in `map` function to extract the heart rates from the health records. While Java 8 provides `map`, it is somewhat clumsy to use and only a handful of CrsB students had been exposed to it by the class (in an optional lab for students who wanted a challenge assignment). A basic Java `for` loop is straightforward for *Data Smoothing*, so we should hardly be surprised that students used it.

Of the 45 CrsB students, 16 produced two *Data Smoothing* solutions with the same high-level structure. This suggests that many CrsB students didn't really understand the idea of multiple program plans just from the single lecture, or perhaps that the alternate plans for *Data Smoothing* were too subtle for many students. Of the strategies provided by the lecture (cleaning, parsing, and truncating), only parsing applies to *Data Smoothing*; if parsing was indeed too hard for students, they would have been left without named strategies to apply to the problem. Instead, they would have to have understood the more general point about different structures allocating tasks differently to traversals of data. The pre-assessment data did not shed light, as all but one student used a *"Single Traversal"* structure to program *Rainfall*. Prior experience is not the explanation either: these 16 students were roughly evenly split between CrsBnvc and CrsBexp. Whether CrsB students would have understood this idea better had they also done ranking problems on the pre-assessment is a question for future studies.

6.3 Preference Ranking of Own Post Solutions

Recall from section 5.4 that students were asked not only to generate two different plans for the solutions, but also to state a preference between the two. Here we study the outcome of this activity from both courses (CrsA and CrsB).

All the CrsA students submitted preferences and mentioned some criteria. Figure 6 shows their preferences for *Earthquake Monitor* solutions. For now, we ignore the colors and look at each bar as a whole. In so doing we notice that students have a significant preference for parsing first, which suggests at least the ability to recognize the lecture's views on structures that better decomposed plans ([☞4]). The *Earthquake Monitor* bars of fig. 3 also show the variety of criteria that the students mentioned.

We now contrast this to CrsB. Figure 7 shows which of their solutions students preferred. Of the 34 students who submitted a solution for *Earthquake Monitor*, all but 5 had at least one *"Single Traversal"* or *"Nested Traversal"* solution, yet half preferred a parsing- or cleaning-based solution. When we now focus on their *criteria*, however, we see something more disturbing. Of the 45 students sampled: only 15 mentioned any criteria at all; 9 didn't submit a ranking; the other 21 just described the implementations (6 of these were from CrsBexp, the rest from CrsBnvc). Among the 15 CrsB students who did describe criteria, code structure and aesthetics came up most often (9 and 10 instances, respectively), while efficiency got only 3 mentions.

The contrast between the courses in students' ability to discuss solutions by attributes is striking, and not readily explained. Neither course had practiced this skill, either explicitly or implicitly that the instructors can recall. Both courses had covered rudimentary big-O prior to the pre-assessment, so students at least had "efficiency" in their vocabulary. Therefore, we do not yet have a proper explanation for these differences ([☞6]).

6.4 Changes in Solution Structures

Changes in the solution structures that students wrote from the pre- to the post-assessment might indicate that the planning lecture had impact. Because students were asked to write two solutions in the post-assessment, it might not be clear which one to compare. However, given that students were asked to rank their two solutions, we believe it is reasonable to compare the structure of the pre-solution with that of the preferred post- one.

We present this comparsion for CrsA, using *Adding Machine* from the pre-assessment and the preferred *Earthquake Monitor* solution from the post-assessment. This is a meaningful comparison due to the similarities in tasks between these two problems: both involve

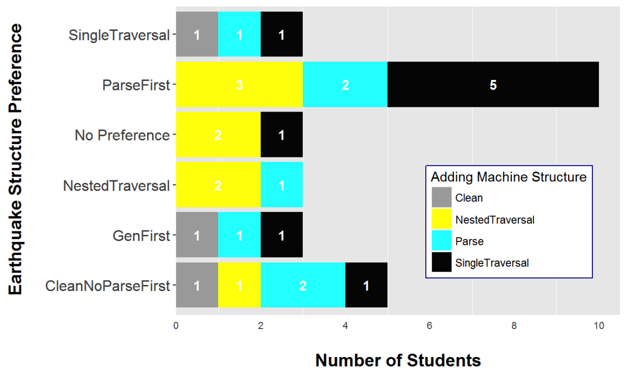

Figure 6: Comparing individual CrsA **students' *Adding Machine* structures (pre) to that of their preferred *Earthquake Monitor* solution (post)**

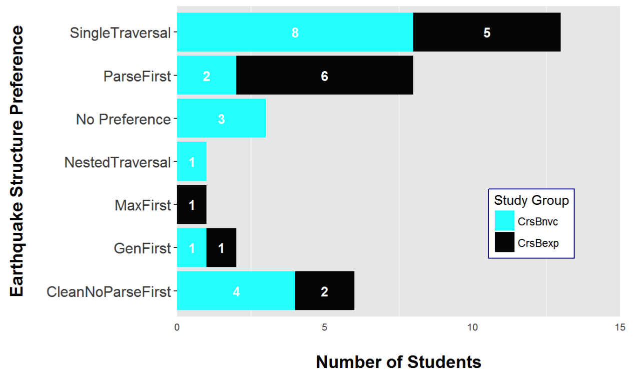

Figure 7: CrsB **students' preferred *Earthquake Monitor* structures (post)**

flattened data, a numeric calculation on subsegments of the data, and consideration of only a subset of the data.

Figure 6 shows this information. Now we can consider the colors. Each row shows a structure for the preferred post- solution binned by their pre- solution structure. The comparisons are per student. Perhaps the most interesting feature of this graph is its lack of a clear internal pattern: students from each *Adding Machine* structure are dispersed across the *Earthquake Monitor* bins, and each *Earthquake Monitor* bin is populated with students from multiple *Adding Machine* structures. Our key takeaway from this graph is that the lecture got CrsA students to think about plans and tradeoffs, with many reconsidering choices that they might have made reflexively during the pre-assessment.

A similar graph for CrsB is not meaningful, since all but one student produced the same structure for *Rainfall* on the pre-assessment. However, we do see diversification in students' preferred solutions in the post-assessment. While *"Single Traversal"* solutions remain the most popular among CrsB students in both the *Earthquake Monitor* and *Data Smoothing* problems on the post-assessment, there is considerable diversity in the *Earthquake Monitor* preferences (which admits more interesting plans). Whether this diversification was caused by the lecture, or by the relatively greater difficulty of *Earthquake Monitor* compared to *Rainfall*, is open to question.

7 THREATS TO VALIDITY

Students' prior programming experience, including the kinds of problems they have been exposed to and in which programming languages, is a significant factor in studies of planning (section 3). We have a rough characterization of our participants' prior experience: students in CrsBexp and CrsA all took courses designed for students with non-trivial programming experience prior to starting at university (for CrsBexp, that course was the one preceding CrsB itself; novice and experienced students feed into a common CrsB as a second course). The nature of that experience could vary significantly across students (in practice, most students had experience in at least Java). More careful accounting of the details of prior background would help refine conclusions in planning studies.

Student motivation to master programmming could affect how seriously they engage in the task of writing two solutions. All students in our study courses were likely interested in majoring in computer science, or at least studying it in some depth. This might give them greater motivation for understanding the planning concepts we discussed, and they might also have more aptitude for computational problem composition and decomposition. It would be interesting to perform similar studies on non-major populations, in particular exploring how much training they require before they appreciate planning. Data on this question will likely evolve in the next few years, given the growing adoption of computing in middle- and high-schools in many countries, which might prepare students for this material.

As noted in section 5.2, we did not use the same questions in both courses. The questions we gave to CrsA reflected the richness and parallel structures across pre and post that we would ideally like to explore, but some of these questions were beyond what the CrsBnvc students would be able to handle based on the problems presented in class (for example, they had not yet done any string manipulation, so *Palindrome* would not have been approachable). The extent to which this is a problem depends a bit on the nature of the conclusions one wishes to draw. Had the study results been positive with all three groups of students and we drawn conclusions about relative performance, the variations in questions would confound the results. As it were, however, the CrsB students did not fare as well, despite having arguably easier questions. In this context, the difference in questions is not as significant.

One reviewer of this work posited that our questions were biased in favor of functional programming. We drew our questions from a collection of planning problems [7] that had been curated by multiple instructors, some of whom vastly prefer functional programming and some who vastly prefer imperative programming. Given this curation, we were confident that our problems were reasonable ones to pose of students working in various programming styles.

8 DISCUSSION AND FUTURE WORK

We now discuss in some depth the many ☞ questions we have raised throughout the paper.

(1) The studies we conducted were roughly near the end of the first semester (in US terminology). If we could wait another semester and get students at the end of the first year, we might find that even those without prior computing are much more sophisticated programmers. It would be especially interesting to see whether, at that point, the CrsBnvc group performs like the CrsA and CrsBexp populations did in our first-semester study.
(2) What if we were to do the same study in an imperative setting? Would students who have primarily been exposed to `for` loops, and not seen higher-level constructs like mapping and filtering, readily grasp the idea of planning? Clearly, exposure to such *constructs* helps students understand planning *concepts*; but while sufficient, is it also necessary? This is a topic that needs further study.
In addition, programming classes may also start changing in flavor to keep pace with languages. Increasingly, imperative and object-oriented languages have adopted some of the basic features of functional programming, such as higher-order functions and higher-order operators (whether in the form of functions like `map` or `filter` or as syntactic constructs such as comprehensions). The pedagogy is also catching up: new (editions of some) Java books provide a thorough discussion of using higher-order functions and functional style [17]. Therefore, it is no longer necessary to switch to a functional language to teach a more functional style of programming. This removes a significant source of friction that introductory course instructors sometimes feel. Nevertheless, this does require adopting a new style of programming, which may be considered a significant intervention in some departments, far removed from our view of it as a "lightweight" one.
(3) What do students already understand about planning before our intervention? Can we phrase our "construct two different plans" task (section 5.4) in a way that meaningfully measures student knowledge and ability?
(4) The fact that students changed their ranking criteria in the post-lecture assessment does not mean they genuinely changed their preference: they may be reflecting what they believe the course staff want to hear. We can probe their true beliefs by giving them significant problems that they can decompose in different ways and checking whether their decompositions match their stated preferences. In turn, creating multi-part exercises where they have to create an initial solution and later modify it—so that issues like maintainability come to the fore—can help reinforce the value of some plans over others.
(5) When are students ready to understand and adopt a planning strategy like parsing? There are many reasons why several CrsBnvc students were unable to use it: maybe they didn't understand it in the first place; maybe they understood it but didn't see its value (especially if they have had no experience building larger systems, it can be hard to see the benefit of an abstraction); or, having passed both hurdles, they may have been unable to implement it.
(6) We find puzzling the contrast between CrsA and CrsB students when it comes to ranking their own solutions. Do the CrsB students not appreciate tradeoffs at all? Do they appreciate them but lack the vocabulary to articulate them? Perhaps they simply did not understand what the problem was asking for?
There is one important factor that may have played a part. The CrsA students had to preference-rank in their pre-assessment, and discussed this during the intervention lecture. Therefore, it is possible they had already had "training" to think about this issue. However, given that they already mentioned several criteria—instead of just describing implementations—in their rankings, this cannot be the whole explanation. Having the CrsB students preference-rank in the pre-assessment would clearly help shed light on this phenomenon.

Going beyond these questions, future studies could include control groups or find ways to determine what kinds of solutions students can imagine as part of the pre-test. Given the relationship between planning and prior exposure to similar problems, control groups would likely need to come through the same sequence of courses as study participants. Naturally, this requires a different intervention design than ours, which covered planning as one of the regular lectures within the participating courses.

9 ACKNOWLEDGMENTS

This research was partially supported by the US NSF under grant 1116539. Reviewers of submitted versions of this paper provided helpful comments on content and organization.

A CODING SOLUTION STRUCTURE

We coded individual solution structures by (a) enumerating the subtasks for each programming problem, and (b) writing a regular expression to capture the clustering and sequencing of subtasks within the solution. We grouped the regular expressions into larger bins based on how they handled the main strategies (parsing, cleaning, truncating) covered in this study. For *Data Smoothing*, for example, the subtasks were (E)xtracting the heart-rate, (S)moothing the data and optionally (P)arsing the data. A code of `P;(E+S)` captures a solution that parses the data then extracts the heart-rates and computes smoothed values in a subsequent traversal. This solution would be classified as *"Parse First"*. Section 5.2 outlined four larger bins that arise for *Adding Machine*; similar terms describe bins for the other problems.

B CODING PREFERENCE CRITERIA

Ranking criteria were processed through open-coding. Four main themes emerged:
(1) **Efficiency**: Mentions runtime, performance (e.g. Big-O), memory use, or efficiency of operations used.
(2) **Structure**: Discusses use of specific operations, constructs, or clusterings of subtasks
(3) **Aesthetics**: Readability, comprehensibility, or reflecting the problem statement in the code; often positive tone.
(4) **Maintainability**: Mentions ability to maintain, debug, or adapt the solution to new data

REFERENCES

[1] Michael E. Caspersen. 2007. *Educating Novices in the Skills of Programming.* Ph.D. Dissertation. University of Aarhus, Denmark.
[2] Francisco Enrique Vicente Castro and Kathi Fisler. 2016. On the Interplay Between Bottom-Up and Datatype-Driven Program Design. In *Proceedings of the ACM Technical Symposium on Computer Science Education.* ACM, 205–210.
[3] Michael de Raadt, Mark Toleman, and Richard Watson. 2004. Training Strategic Problem Solvers. *SIGCSE Bull.* 36, 2 (June 2004), 48–51.
[4] Michael de Raadt, Richard Watson, and Mark Toleman. 2009. Teaching and Assessing Programming Strategies Explicitly. In *Proceedings of the Australasian Computing Education Conference.* Australian Computer Society, Inc., 45–54.

[5] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2001. *How to Design Programs*. MIT Press. http://www.htdp.org/

[6] Kathi Fisler. 2014. The Recurring Rainfall Problem. In *Proceedings of the International Conference on Computing Education Research*. ACM, 35–42.

[7] Kathi Fisler, Shriram Krishnamurthi, and Janet Siegmund. 2016. Modernizing Plan-Composition Studies. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. ACM, 211–216.

[8] David Ginat. 2009. Interleaved Pattern Composition and Scaffolded Learning. In *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education*. ACM, 109–113.

[9] S.P. Marshall. 1995. *Schemas in Problem Solving*. Cambridge University Press. https://books.google.com/books?id=iKSYrsXe6f0C

[10] Orna Muller, David Ginat, and Bruria Haberman. 2007. Pattern-oriented Instruction and Its Influence on Problem Decomposition and Solution Construction. In *Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education*. ACM, 151–155.

[11] A. Newell, P.S. Rosenbloom, and J.E. Laird. 1989. Symbolic Architectures for Cognition. In *Foundations of Cognitive Science*. MIT Press, 93–131.

[12] Peter L. Pirolli and John R. Anderson. 1985. The Role of Learning from Examples in the Acquisition of Recursive Programming Skills. *Canadian Journal of Psychology/Revue canadienne de psychologie* 39, 2 (1985), 240–272.

[13] Peter L. Pirolli, John R. Anderson, and Robert G. Farrell. 1984. *Learning to program recursion*. 277–280.

[14] Ron Porter and Paul Calder. 2003. A Pattern-based Problem-solving Process for Novice Programmers. In *Proceedings of the Australasian Computing Education Conference*. Australian Computer Society, Inc., 231–238.

[15] Robert S. Rist. 1989. Schema Creation in Programming. *Cognitive Science* (1989), 389–414.

[16] Otto Seppala, Petri Ihantola, Essi Isohanni, Juha Sorva, and Arto Vihavainen. 2015. Do We Know How Difficult the Rainfall Problem is?. In *Proceedings of the Koli Calling Conference on Computing Education Research*. ACM, 87–96.

[17] Peter Sestoft. 2016. *Java Precisely* (third ed.). MIT Press.

[18] Simon. 2013. Soloway's Rainfall Problem Has Become Harder. *Learning and Teaching in Computing and Engineering* (2013), 130–135.

[19] E. Soloway. 1986. Learning to Program = Learning to Construct Mechanisms and Explanations. *Commun. ACM* 29, 9 (Sept. 1986), 850–858.

[20] James C. Spohrer and Elliot Soloway. 1989. *Simulating Student Programmers*. Morgan Kaufmann Publishers Inc., 543–549.